Cache算法设计与实现

1. **总体思路**

在本次实验要求中，给出了Cache大小（256KB），Block大小（16B），内存地址的位数为64位（数据从外部文件中逐行读取），Cache的替换算法为LRU算法，需要分别使用直接映射DM、组关联SA方法、MRU预测方法和Multi-column预测方法模拟Cache行为，并且计算对应的命中率。在本算法设计中，MRU和Multi-column都属于提高Cache的First Hit，加快Cache命中的预测算法，最终的Cache替换策略都为LRU，因此并不会改变Cache总的命中率。

本次实验使用C++语言在VSCODE编译环境中进行算法实现，主要设计思路如图1所示：

图1 Cache算法设计总体思路

主程序（Main Process）负责从外部读取TRACE文件，解析地址中的Tag、Set与Offset, 并与Cache中的组号进行映射（b = B mod C，其中b为cache中块号，B为主存中的块号，C为cache中块容量），然后选择调用LRU、MRU或Multi-column算法对CACHE进行访问或替换，同时将命中结果输出到终端，将Miss的记录保存到外部文件中。

LRU、MRU和Multi-column算法可以封装成为一个类（CloveCache类），用于对一块地址空间进行访问策略和换入换出管理。类的底层使用LinkList、Map、Vector等数据结构进行组织，便于完成不同的功能。

Cache使用数组进行模拟。每个数组单元代表Cache中的一个组，组内采用类（CloveCache类）进行管理。CloveCache类可以完成Cache组内数据的MRU预测访问、 Multi-column预测访问以及LRU换入换出等管理工作。

1. **算法设计**
2. Cache设计

Cache采用数组进行模拟。为了设计的统一，将直接映射认为是1路组相联。设Cache大小为CACHE\_SIZE，块大小为BLOCK\_SIZE，Cache组数为CACHE\_SET，Cache组内路数为CACHE\_WAY，对于Cache大小为256KB，Block大小为16B，内存地址的位数为64位来说，则有：

const int MEM\_ADDR\_LEN = 64;

const int CACHE\_SIZE = 256 \* 1024;

const int BLOCK\_SIZE = 16;

const int CACHE\_WAY = 4; //1: DM 2: 2-way 4: 4-way……

const int CACHE\_SET = CACHE\_SIZE / BLOCK\_SIZE / CACHE\_WAY; //e.g. 2^14

//for a 64-bit memory addr, for it's block size is 16B, so offset is:4

const int offset = 4;

const int index = log(CACHE\_SET)/log(2);// the cache lines, e.g. 14

const int preOffset = log(CACHE\_WAY)/log(2);

const int tag = MEM\_ADDR\_LEN - index - offset; // e.g. 64- 14 - 4

Cache组内用CloveCache类结构，则Cache表示为：

CloveCache\* cache[CACHE\_SET]

Cache每个组的初始化需要传递组内的相联路数，表示如下：

cache[i] = new CloveCache(CACHE\_WAY)

1. CloveCache类

该类用于对Cache中的一个组内部数据结构进行定义与管理。如图2所示：



图2 Cache组内数据结构

CloveCache类使用链表结构表示LUR队列，其中Head表示mru位置，Tail表示lru位置，即新数据从Head挂入，将要被替换出去的数据从Tail进行移除。Cache中每一路采用双向链表存储地址的Tag信息，双向链表ListNode设计如下：

typedef struct ListNode

{

    ulong64 value;

    struct ListNode \*pPre;

    struct ListNode \*pNext;

} ListNode;

为了加快索引，CloveCache类使用Map结构模拟Cache Set内Ways的存储，Multi-column算法中使用Vector向量存储，对于MRU预测算法，采用一个整型值进行模拟，数据结构定义如下：

int m\_capacity; //capacity of LRU

ListNode \*m\_pHead; //head pointer, store the newest data

ListNode \*m\_pTail; //tail pointer, the LRU data

map<int, ListNode \*> m\_mp;

int m\_MRUPre; // mru for predicting, store the cache-way index

vector<vector<int>> m\_selectedLocation;

CloveCache类中定义了LRU算法、MRU算法和Multi-column的Full indexing以及Partial indexing方法。

int LRU(ulong64 value);

int MruPre(ulong64 value);

int MccPre(ulong64 value, bool beFullIndex = true);

1. LRU算法

LRU的算法流程如图3所示：



图3 LRU算法设计流程图

数据Value1到来后，首先在Cache的Map中查找数据是否存在，如果存在则直接命中，如果不存在，则根据Map当前容量的大小选择是否需要替换一块最近最少使用的数据出去。

针对Map+LinkList的数据结构，LRU在进行换入换出以及调整MRU（注意：是该节点的链表属性，不是Map中的位置）时主要进行三个操作：Tail移除操作（Remove）、Head插入操作（SetHead）以及Map赋值与记录操作。

**1．Remove方法**

Remove方法是将链表中特定Node从节点中移除，但并不删除该数据。主要有3种情况：

1）如果是头节点head，则将head移除，head的next成为head;

m\_pHead = pNode->pNext;

2）如果是尾节点tail，则将tail移除，tail的pre成为tail;

m\_pTail = pNode->pPre;

3）如果是中间某节点pnode，则将pnode移除，pnode的pre和next进行建链。

pNode->pPre->pNext = pNode->pNext;

pNode->pNext->pPre = pNode->pPre;

**2．SetHead方法**

SetHead方法直接将指定节点插入链表中Head的前面，并根据LinkList中Head和Tail的属性尽心指针定位。主要的代码如下：

pNode->pNext = m\_pHead;pNode->pPre = NULL;

m\_pHead->pPre = pNode;m\_pHead = pNode;

因此LRU中的换入换出以及调整MRU位置只可能是下面三种情况：

1. 命中。将当前命中位置设置为MRU位置。

ListNode \*Node = it->second;

Remove(Node);SetHead(Node);

mruKey = it->first; // record the most resent access cache-way

2）缺失且链表满。将LRU位置（LinkList中Tail位置）数据移除，并在链表Head插入新的数据节点。此时由于是新换入了数据，因此需要同步修改map中对应位置的值。

it = find\_if(m\_mp.begin(), m\_mp.end(), MapValueFinder(m\_pTail->value));

Remove(m\_pTail);SetHead(NewNode);

m\_mp[it->first] = NewNode; //new data take place the old data

mruKey = it->first;

3）缺失且链表未满。只需在map中空位置处插入新的节点，并将该节点置为MRU位置。

SetHead(NewNode);

mruKey = m\_index; // record the most resent access cache-way

m\_mp[m\_index++] = NewNode;

1. MRU算法

MRU预测算法在首次访问Cache即命中的情况下能够加快Cache的访问时间，它是在LRU的基础上设定一个MRU变量用于存储最近最新访问的数据位置。算法主要点在于每次命中和换入换出时记录当前的最新位置，并将其设置为MRU位置。

mruKey = it->first; 或者 mruKey = m\_index;

每次从Map中查找数据前，都先检查MRU中对应的位置，如果刚好命中，则返回First Hit（return 1），不用更新MRU数值（因为命中的本来就是MRU），否则使用LRU算法。

if (it != m\_mp.end() && m\_mp[m\_MRUPre]->value == value)

return 1;

return LRU(value);

MRU的算法流程如图4所示：



图4 MRU算法设计流程图

1. Multi-column算法

相比较与MRU，Multi-column能够加大first hits的命中率，并且减少non-first-hits的时间延迟，在first hit命中时相当于一次直接映射，在增加少量硬件的情况下这是非常值得和划算的。本次算法模拟的主要思想是增加一个二维向量用于模拟Selected Locations的硬件结构，算法设计主要考虑以下方面：

1. 使用Tag的最后n位（n= log2(CACHE\_WAY)）直接对应Cache中的Major Location。Major Location是由Tag的最后n位决定的，从Mem中新调度的数据以及刚刚访问过的数据都应该放到对用的Major Location位置上；
2. First hit通过map的key索引直接访问数据，相当于模拟一次直接映射；
3. Multiple blocks can be direct -mapped to the same major location, but only MRU is the major；
4. The non-MRU blocks are stored in other empty MRU blocks are stored in other locations in the set: Selected Locations；
5. If “other locations” are used for their own major locations, there will be no space for selected ones；
6. Swap: A block in selected selected location is swapped to major location after a new block is loaded in from memory。

Multi-column的主算法流程（包括Full indexing和Partial indexing）如图5所示：



图5 Multi-column主算法设计流程图

算法流程描述如下：

1. 根据Cache组内路数的大小获取Multi-column的Major Location值；

int predictedWay = value1 % m\_capacity;

1. 根据predictedWay（PreOffset）值直接映射到Map中匹配Major Location，如果该位置为空，则将value放置在当前位置，设置该位置为MRU，并在Selected Locations中设置主位置标志为-1（用-1标志主位置，便于后面算法进行swap），返回miss(0)；如果不为空，则转向3）；

       m\_selectedLocation[predictedWay][predictedWay] = -1;

1. 如果是First hit，则将当前位置设置为MRU，并返回first hit(1)；否则转向4；
2. 如果是Full indexing，则转向5），Partial indexing则转向9）；

if (beFullIndex)

   return SearchSelectedLocationsFullIndexing(value, predictedWay);

return SearchSelectedLocationsPartialIndexing(value, predictedWay);

1. 根据Selected Locations进行non-first-hit，只需要检查值为1的位置，如果命中，则设置该位置为MRU，并将它和Major Location互换，返回2；否则转向6）；

if (m\_selectedLocation[predictedWay][i] == 1) // selected location i

   if (m\_mp[i] && m\_mp[i]->value == value) //hit

   { ListNode \*Node = m\_mp[i];

       Remove(Node); SetHead(Node);  Swap(i, predictedWay);

return 2;

}

}

1. 判断当前Map是否已满，如果未满，则转向7；如果已满，则首先根据LRU策略寻找value1要替换的位置，并将该节点（是该节点的链表属性，不是Map中的位置）设置为MRU，然后将该节点与Major Location中的节点互换位置。互换位置前需要将自己所在Way以及其他Way中的Selected Locations标记进行改变，遵循的原则如下：
2. 如果是自己所在的Way，则将Major Location在Selected Locations中的标志取反设置到Value当前所在位置的Selected Locations中，并将Major Location位重新设置为-1。这是因为如果原来Major Location位置的标记为0，则说明该位置已被其他数据占据，互换位置后对应位的标记也应该为0；如果Major Location位置的标记为-1，则说明该位置为主位置，互换位置后对应位的标记应该为1。
3. 如果不是自己所在的Way，则将Major Location在Selected Locations中的标志直接设置到value1当前所在位置的Selected Locations中（告诉对方已将其Selected Location移到了其他位置上），并将Major Location位重新设置为0（告诉对方该位置我已收回）；

if (i != predictedWay)

{

   m\_selectedLoc[i][it->first] = m\_selectedLoc[i][predictedWay];

   m\_selectedLoc[i][predictedWay] = 0;

}

else

{

   m\_selectedLoc[i][it->first] = -m\_selectedLoc[i][predictedWay];

   m\_selectedLoc[i][predictedWay] = -1;

}

1. 寻找空位置放置该value1，并将该节点设置为MRU，然后将该节点与Major Location中的节点互换位置。遵循的原则与6）相同；
2. 6）或7）完成后返回0。
3. 根据Selected Locations进行non-first-hit，只需要检查值为1的位置，并且只检查一次（一个Major Location和一个Selected Location已经能够大程度上加大first hits的命中率，并且减少non-first-hits的时间延迟，是一种tradeoff策略）。如果命中，则设置该节点为MRU，并将它和Major Location互换，返回2；否则转向10）；
4. 使用LRU方法在Map中其他位置寻找匹配（此时在查询时间上不再具有优势），如果命中，则设置该节点为MRU，并将它和Major Location互换，返回3；否则转向11）；
5. 该策略与6）基本一致，只在Selected Location的设置上有所不同。判断当前Map是否已满，如果未满，则转向12）。与6）的主要区别在于需要判断Selected Locations中是否已经有一个Location为1，如果已存在，则放置的新位置的Selected Location需要设置为0。完成后返回0。

//判断是不是已经有一个selected pos了,或者当前替换出去的就是这个标识为1的位置

if(selectedPos < 0 || it->first == selectedPos)

{

  m\_selectedLoc[i][it->first] = -m\_selectedLoc[i][predictedWay];

  m\_selectedLoc [i][predictedWay] = -1;

}

else//在其他位置已经存在一个是1的位置了

{

  m\_selectedLoc[i][it->first] = 0;

  m\_selectedLoca[i][predictedWay] = -1;//将自己的MRU位置标记为-1

}

1. 寻找空位置放置该value1，并将该节点设置为MRU，然后将该节点与Major Location中的节点互换位置。遵循的原则与11同。完成后返回0。
2. **实验结果**
3. **分析总结**